Title

Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff.

Authors

Bhattacherjee, Souvik
Chavan, Amit
Huang, Silu
et al.

Publication Date

2015-08-01

DOI

10.14778/2824032.2824035

Peer reviewed

Published in final edited form as:

Proceedings VLDB Endowment. 2015 August ; 8(12): 1346–1357. doi:10.14778/2824032.2824035.

Principles of Dataset Versioning: Exploring the Recreation/ Storage Tradeoff

Souvik Bhattacherjee[1], **Amit Chavan**[1], **Silu Huang**[2], **Amol Deshpande**[1], and **Aditya Parameswaran**[2]

[1]University of Maryland, College Park

[2]University of Illinois, Urbana-Champaign

Abstract

The relative ease of collaborative data science and analysis has led to a proliferation of many thousands or millions of *versions* of the same datasets in many scientific and commercial domains, acquired or constructed at various stages of data analysis across many users, and often over long periods of time. Managing, storing, and recreating these dataset versions is a non-trivial task. The fundamental challenge here is the *storage-recreation trade-off*: the more storage we use, the faster it is to recreate or retrieve versions, while the less storage we use, the slower it is to recreate or retrieve versions. Despite the fundamental nature of this problem, there has been a surprisingly little amount of work on it. In this paper, we study this trade-off in a principled manner: we formulate six problems under various settings, trading off these quantities in various ways, demonstrate that most of the problems are intractable, and propose a suite of inexpensive heuristics drawing from techniques in delay-constrained scheduling, and spanning tree literature, to solve these problems. We have built a prototype version management system, that aims to serve as a foundation to our DataHub system for facilitating collaborative data science. We demonstrate, via extensive experiments, that our proposed heuristics provide efficient solutions in practical dataset versioning scenarios.

1. INTRODUCTION

The massive quantities of data being generated every day, and the ease of collaborative data analysis and data science have led to severe issues in management and retrieval of datasets. We motivate our work with two concrete example scenarios.

- [Intermediate Result Datasets] For most organizations dealing with large volumes of diverse datasets, a common scenario is that many datasets are repeatedly analyzed in slightly different ways, with the intermediate results stored for future use. Often, we find that the intermediate results are the same across many pipelines (e.g., a *PageRank* computation on the Web graph is often part of a multi-step workflow). Often times, the datasets being analyzed might be slightly different (e.g., results of simple transformations or cleaning operations,

or small updates), but are still stored in their entirety. There is currently no way of reducing the amount of stored data in such a scenario: there is massive redundancy and duplication (this was corroborated by our discussions with a large software company), and often the computation required to recompute a given version from another one is small enough to not merit storing a new version.

- [Data Science Dataset Versions] In our conversations with a computational biology group, we found that every time a data scientist wishes to work on a dataset, they make a private copy, perform modifications via cleansing, normalization, adding new fields or rows, and then store these modified versions back to a folder shared across the entire group. Once again there is massive redundancy and duplication across these copies, and there is a need to minimize these storage costs while keeping these versions easily retrievable.

In such scenarios and many others, it is essential to keep track of versions of datasets and be able to recreate them on demand; and at the same time, it is essential to minimize the storage costs by reducing redundancy and duplication. The ability to manage a large number of datasets, their versions, and derived datasets, is a key foundational piece of a system we are building for facilitating collaborative data science, called DataHub [12]. DataHub enables users to keep track of datasets and their versions, represented in the form of a directed *version graph* that encodes derivation relationships, and to retrieve one or more of the versions for analysis.

In this paper, we focus on the problem of trading off storage costs and recreation costs in a principled fashion. Specifically, the problem we address in this paper is: given a collection of datasets as well as (possibly) a directed version graph connecting them, minimize the overall storage for storing the datasets and the recreation costs for retrieving them. The two goals conflict with each other — minimizing storage cost typically leads to increased recreation costs and vice versa. We illustrate this trade-off via an example.

Example 1: Figure 1(i) displays a version graph, indicating the derivation relationships among 5 versions. Let V_1 be the original dataset. Say there are two teams collaborating on this dataset: team 1 modifies V_1 to derive V_2, while team 2 modifies V_1 to derive V_3. Then, V_2 and V_3 are merged and give V_5. As presented in Figure 1, V_1 is associated with ⟨10000, 10000⟩, indicating that V_1's storage cost and recreation cost are both 10000 when stored in its entirety (we note that these two are typically measured in different units – see the second challenge below); the edge ($V_1 \rightarrow V_3$) is annotated with ⟨1000, 3000⟩, where 1000 is the storage cost for V_3 when stored as the modification from V_1 (we call this the delta of V_3 from V_1) and 3000 is the recreation cost for V_3 given V_1, i.e, the time taken to recreate V_3 given that V_1 has already been recreated.

One naive solution to store these datasets would be to store all of them in their entirety (Figure 1 (ii)). In this case, each version can be retrieved directly but the total storage cost is rather large, i.e., 10000 + 10100 + 9700 + 9800 + 10120 = 49720. At the other extreme, only one version is stored in its entirety while other versions are stored as modifications or deltas

to that version, as shown in Figure 1 (iii). The total storage cost here is much smaller (10000 + 200 + 1000 + 50 + 200 = 11450), but the recreation cost is large for V_2, V_3, V_4 and V_5. For instance, the path $\{(V_1 \rightarrow V_3 \rightarrow V_5)\}$ needs to be accessed in order to retrieve V_5 and the recreation cost is 10000 + 3000 + 550 = 13550 > 10120.

Figure 1 (iv) shows an intermediate solution that trades off increased storage for reduced recreation costs for some version. Here we store versions V_1 and V_3 in their entirety and store modifications to other versions. This solution also exhibits higher storage cost than solution (ii) but lower than (iii), and still results in significantly reduced retrieval costs for versions V_3 and V_5 over (ii).

Despite the fundamental nature of the storage-retrieval problem, there is surprisingly little prior work on formally analyzing this trade-off and on designing techniques for identifying effective storage solutions for a given collection of datasets. Version Control Systems (VCS) like Git, SVN, or Mercurial, despite their popularity, use fairly simple algorithms underneath, and are known to have significant limitations when managing large datasets [1, 2]. Much of the prior work in literature focuses on a linear chain of versions, or on minimizing the storage cost while ignoring the recreation cost (we discuss the related work in more detail in Section 6).

In this paper, we initiate a formal study of the problem of deciding how to jointly store a collection of dataset versions, provided along with a version or derivation graph. Aside from being able to handle the scale, both in terms of dataset sizes and the number of versions, there are several other considerations that make this problem challenging.

- Different application scenarios and constraints lead to many variations on the basic theme of balancing storage and recreation cost (see Table 1). The variations arise both out of different ways to reconcile the conflicting optimization goals, as well as because of the variations in how the differences between versions are stored and how versions are reconstructed. For example, some mechanisms for constructing differences between versions lead to symmetric differences (either version can be recreated from the other version)— we call this the *undirected* case. The scenario with asymmetric, one-way differences is referred to as *directed* case.
- Similarly, the relationship between storage and recreation costs leads to significant variations across different settings. In some cases the recreation cost is proportional to the storage cost (e.g., if the system bottleneck lies in the I/O cost or network communication), but that may not be true when the system bottleneck is CPU computation. This is especially true for sophisticated differencing mechanisms where a compact derivation procedure might be known to generate one dataset from another.
- Another critical issue is that computing deltas for all pairs of versions is typically not feasible. Relying purely on the version graph may not be sufficient and significant redundancies across datasets may be missed.

- Further, in many cases, we may have information about relative *access frequencies* indicating the relative likelihood of retrieving different datasets. Several baseline algorithms for solving this problem cannot be easily adapted to incorporate such access frequencies.

We note that the problem described thus far is inherently "online" in that new datasets and versions are typically being created continuously and are being added to the system. In this paper, we focus on the static, off-line version of this problem and focus on formally and completely understanding that version. We plan to address the online version of the problem in the future. The key contributions of this work are as follows.

- We formally define and analyze the dataset versioning problem and consider several variations of the problem that trade off storage cost and recreation cost in different manners, under different assumptions about the differencing mechanisms and recreation costs (Section 2). Table 1 summarizes the problems and our results. We show that most of the variations of this problem are NP-Hard (Section 3).
- We provide two light-weight heuristics: one, when there is a constraint on average recreation cost, and one when there is a constraint on maximum recreation cost; we also show how we can adapt a prior solution for balancing minimum-spanning trees and shortest path trees for undirected graphs (Section 4).
- We have built a prototype system where we implement the proposed algorithms. We present an extensive experimental evaluation of these algorithms over several synthetic and real-world workloads demonstrating the effectiveness of our algorithms at handling large problem sizes (Section 5).

2. PROBLEM OVERVIEW

In this section, we first introduce essential notations and then present the various problem formulations. We then present a mapping of the basic problem to a graph-theoretic problem, and also describe an integer linear program to solve the problem optimally.

2.1 Essential Notations and Preliminaries

Version Graph—We let $\mathcal{V} = \{V_i\}, i = 1, \ldots, n$ be a collection of versions. The derivation relationships between versions are represented or captured in the form of a *version graph*: $g(\mathcal{V}, \mathcal{E})$. A directed edge from V_i to V_j in $g(\mathcal{V}, \mathcal{E})$ represents that V_j was derived from V_i (either through an update operation, or through an explicit transformation). Since branching and merging are permitted in DataHub (admitting collaborative data science), g is a DAG (directed acyclic graph) instead of a linear chain. For example, Figure 1 represents a version graph g, where V_2 and V_3 are derived from V_1 separately, and then merged to form V_5.

Storage and Recreation—Given a collection of versions $\mathcal{V}$, we need to reason about the *storage cost*, i.e., the space required to store the versions, and the *recreation cost*, i.e., the time taken to recreate or retrieve the versions. For a version V_i, we can either:

- Store V_i in its entirety: in this case, we denote the storage required to record version V_i fully by $\Delta_{i,i}$. The recreation cost in this case is the time needed to retrieve this recorded version; we denote that by $\Phi_{i,i}$. A version that is stored in its entirety is said to be *materialized*.
- Store a "delta" from V_j: in this case, we do not store V_i fully; we instead store its modifications from another version V_j. For example, we could record that V_i is just V_j but with the 50th tuple deleted. We refer to the information needed to construct version V_i from version V_j as the *delta* from V_j to V_i. The algorithm giving us the delta is called a *differencing algorithm*. The storage cost for recording modifications from V_j, i.e., the size the delta, is denoted by $\Delta_{j,i}$. The recreation cost is the time needed to recreate the recorded version given that V_j has been recreated; this is denoted by $\Phi_{j,i}$.

Thus the storage and recreation costs can be represented using two matrices Δ and Φ: the entries along the diagonal represent the costs for the materialized versions, while the off-diagonal entries represent the costs for deltas. From this point forward, we focus our attention on these matrices: they capture all the relevant information about the versions for managing and retrieving them.

Delta Variants—Notice that by changing the differencing algorithm, we can produce deltas of various types:

- for text files, UNIX-style diffs, i.e., line-by-line modifications between versions, are commonly used;
- we could have a listing of a program, script, SQL query, or command that generates version V_i from V_j;
- for some types of data, an XOR between the two versions can be an appropriate delta; and
- for tabular data (e.g., relational tables), recording the differences at the cell level is yet another type of delta.

Furthermore, the deltas could be stored compressed or uncompressed. The various delta variants lead to various dimensions of problem that we will describe subsequently.

The reader may be wondering why we need to reason about two matrices Δ and Φ. In some cases, the two may be proportional to each other (e.g., if we are using uncompressed UNIX-style diffs). But in many cases, the storage cost of a delta and the recreation cost of applying that delta can be very different from each other, especially if the deltas are stored in a compressed fashion. Furthermore, while the storage cost is more straightforward to account for in that it is proportional to the bytes required to store the deltas between versions, recreation cost is more complicated: it could depend on the network bandwidth (if versions or deltas are stored remotely), the I/O bandwidth, and the computation costs (e.g., if decompression or running of a script is needed).

Example 2: Figure 2 shows the matrices Δ and Φ based on version graph in Figure 1. The annotation associated with the edge (V_i, V_j) in Figure 1 is essentially $\langle \Delta_{i,j}, \Phi_{i,j} \rangle$, whereas the vertex annotation for V_i is $\langle \Delta_{i,i}, \Phi_{i,i} \rangle$. If there is no edge from V_i to V_j in the version graph, we have two choices: we can either set the corresponding Δ and Φ entries to "−" (unknown) (as shown in the figure), or we can explicitly compute the values of those entries (by running a differencing algorithm). For instance, $\Delta_{3,2} = 1100$ and $\Phi_{3,2} = 3200$ are computed explicitly in the figure (the specific numbers reported here are fictitious and not the result of running any specific algorithm).

Discussion—Before moving on to formally defining the basic optimization problem, we note several complications that present unique challenges in this scenario.

- *Revealing entries in the matrix:* Ideally, we would like to compute all pairwise Δ and Φ entries, so that we do not miss any significant redundancies among versions that are far from each other in the version graph. However when the number of versions, denoted n, is large, computing all those entries can be very expensive (and typically infeasible), since this means computing deltas between all pairs of versions. Thus, we must reason with incomplete Δ and Φ matrices. Given a version graph g, one option is to restrict our deltas to correspond to actual edges in the version graph; another option is to restrict our deltas to be between "close by" versions, with the understanding that versions close to each other in the version graph are more likely to be similar. Prior work has also suggested mechanisms (e.g., based on hashing) to find versions that are close to each other [18]. We assume that some mechanism to choose which deltas to reveal is provided to us.
- *Multiple "delta" mechanisms:* Given a pair of versions (V_i, V_j), there could be many ways of maintaining a delta between them, with different $\Delta_{i,j}$, $\Phi_{i,j}$ costs. For example, we can store a program used to derive V_j from V_i, which could take longer to run (i.e., the recreation cost is higher) but is more compact (i.e., storage cost is lower), or explicitly store the UNIX-style diffs between the two versions, with lower recreation costs but higher storage costs. For simplicity, we pick one delta mechanism: thus the matrices Δ, Φ just have one entry per (i, j) pair. Our techniques also apply to the more general scenario with small modifications.
- *Branches:* Both branching and merging are common in collaborative analysis, making the version graph a directed acyclic graph. In this paper, we assume each version is either stored in its entirety or stored as a delta from a single other version, even if it is derived from two different datasets. Although it may be more efficient to allow a version to be stored as a delta from two other versions in some cases, representing such a storage solution requires more complex constructs and both the problems of finding an optimal storage solution for a given problem instance and retrieving a specific version become much more complicated. We plan to further study such solutions in future.

Matrix Properties and Problem Dimensions—The storage cost matrix Δ may be symmetric or asymmetric depending on the specific differencing mechanism used for

constructing deltas. For example, the XOR differencing function results in a symmetric Δ matrix since the delta from a version V_i to V_j is identical to the delta from V_j to V_i. UNIX-style diffs where line-by-line modifications are listed can either be two-way (symmetric) or one-way (asymmetric). The asymmetry may be quite large. For instance, it may be possible to represent the delta from V_i to V_j using a command like: *delete all tuples with age > 60*, very compactly. However, the reverse delta from V_j to V_i is likely to be quite large, since all the tuples that were deleted from V_i would be a part of that delta. In this paper, we consider both these scenarios. We refer to the scenario where Δ is symmetric and Δ is asymmetric as the undirected case and directed case, respectively.

A second issue is the relationship between Φ and Δ. In many scenarios, it may be reasonable to assume that Φ is proportional to Δ. This is generally true for deltas that contain detailed line-by-line or cell-by-cell differences. It is also true if the system bottleneck is network communication or I/O cost. In a large number of cases, however, it may be more appropriate to treat them as independent quantities with no overt or known relationship. For the proportional case, we assume that the proportionality constant is 1 (i.e., $\Phi = \Delta$); the problem statements, algorithms and guarantees are unaffected by having a constant proportionality factor. The other case is denoted by $\Phi \neq \Delta$.

This leads us to identify three distinct cases with significantly diverse properties: (1) **Scenario 1**: Undirected case, $\Phi = \Delta$; (2) **Scenario 2**: Directed case, $\Phi = \Delta$; and (3) **Scenario 3**: Directed case, $\Phi \neq \Delta$.

Objective and Optimization Metrics—Given Δ, Φ, our goal is to find a good storage solution, i.e., we need to decide which versions to materialize and which versions to store as deltas from other versions. Let $\mathcal{P} = \{(i_1, j_1), (i_2, j_2), \ldots\}$ denote a storage solution. $i_k = j_k$ indicates that the version V_{i_k} is materialized (i.e., stored explicitly in its entirety), whereas a pair (i_k, j_k), $i_k \neq j_k$ indicates that we store a delta from V_{i_k} to V_{j_k}.

We require any solution we consider to be a *valid* solution, where it is possible to reconstruct any of the original versions. More formally, $\mathcal{P}$ is considered a *valid* solution if and only if for every version V_i, there exists a sequence of distinct versions $V_{l_1}, \ldots, V_{l_k} = V_i$ such that (i_{l_1}, i_{l_1}), (i_{l_1}, i_{l_2}), $(i_{l_2}, i_{l_3}), \ldots, (i_{l_{k-1}}, i_{l_k})$ are contained in $\mathcal{P}$ (in other words, there is a version V_{l_1} that can be materialized and can be used to recreate V_i through a chain of deltas).

We can now formally define the optimization goals:

- *Total Storage Cost* (denoted $\mathcal{C}$): The total storage cost for a solution $\mathcal{P}$ is simply the storage cost necessary to store all the materialized versions and the deltas: $\mathcal{C} = \Sigma_{(i,j)\in\mathcal{P}} \Delta_{i,j}$.
- *Recreation Cost for* V_i (denoted $\mathcal{R}_i$): Let $V_{l_1}, \ldots, V_{l_k} = V_i$ denote a sequence that can be used to reconstruct V_i. The cost of recreating V_i using that sequence is: $\Phi_{l_1,l_1} + \Phi_{l_1,l_2} + \ldots + \Phi_{l_{k-1},l_k}$. The recreation cost for V_i is the minimum of these quantities over all sequences that can be used to recreate V_i.

Problem Formulations—We now state the problem formulations that we consider in this paper, starting with two base cases that represent two extreme points in the spectrum of possible problems.

Problem 1 (Minimizing Storage): Given Δ, Φ, find a valid solution $\mathcal{P}$ such that $\mathscr{C}$ is minimized.

Problem 2 (Minimizing Recreation): GivenΔ, Φ, identify a valid solution $\mathcal{P}$ such that ∀i, R_i is minimized.

The above two formulations minimize either the storage cost or the recreation cost, without worrying about the other. It may appear that the second formulation is not well-defined and we should instead aim to minimize the average recreation cost across all versions. However, the (simple) solution that minimizes average recreation cost also naturally minimizes $\mathcal{R}_i$ for each version.

In the next two formulations, we want to minimize (a) the sum of recreation costs over all versions ($\Sigma_i \mathcal{R}_i$), (b) the max recreation cost across all versions ($\max_i \mathcal{R}_i$), under the constraint that total storage cost $\mathscr{C}$ is smaller than some threshold β. These problems are relevant when the storage budget is limited.

Problem 3 (MinSum Recreation): GivenΔ; Φ and a threshold β, identify $\mathcal{P}$ such that $\mathscr{C} \leq \beta$, and$\Sigma_i \mathcal{R}_i$ is minimized.

Problem 4 (MinMax Recreation): GivenΔ, Φ and a threshold β, identify $\mathcal{P}$ such that $\mathscr{C} \leq \beta$, and $\max_i \mathcal{R}_i$ is minimized.

The next two formulations seek to instead minimize the total storage cost $\mathscr{C}$ given a constraint on the sum of recreation costs or max recreation cost. These problems are relevant when we want to reduce the storage cost, but must satisfy some constraints on the recreation costs.

Problem 5 (Minimizing Storage(Sum Recreation)): Given Δ, Φ and a threshold θ, identify $\mathcal{P}$ such that $\Sigma_i \mathcal{R}_i \leq \theta$, and $\mathscr{C}$ is minimized.

Problem 6 (Minimizing Storage(Max Recreation)): *Given* Δ, Φ *and a threshold θ, identify* $\mathcal{P}$ *such that* $\max_i \mathcal{R}_i \leq \theta$, *and* $\mathscr{C}$ *is minimized.*

2.2 Mapping to Graph Formulation

In this section, we'll map our problem into a graph problem, that will help us to adopt and modify algorithms from well-studied problems such as minimum spanning tree construction and delay-constrained scheduling. Given the matrices Δ and Φ, we can construct a directed, edge-weighted graph $G = (V, E)$ representing the relationship among different versions as follows. For each version V_i, we create a vertex V_i in G. In addition, we create a dummy vertex V_0 in G. For each V_i, we add an edge $V_0 \rightarrow V_i$, and assign its edge-weight as a tuple $\langle \Delta_{i,i}, \Phi_{i,i} \rangle$. Next, for each $\Delta_{i,j} \neq \infty$, we add an edge $V_i \rightarrow V_j$ with edge-weight $\langle \Delta_{i,j}, \Phi_{i,j} \rangle$.

The resulting graph G is similar to the original version graph, but with several important differences. An edge in the version graph indicates a derivation relationship, whereas an edge in G simply indicates that it is possible to recreate the target version using the source version and the associated edge delta (in fact, ideally G is a complete graph). Unlike the version graph, G may contain cycles, and it also contains the special dummy vertex V_0. Additionally, in the version graph, if a version V_i has multiple in-edges, it is the result of a user/application merging changes from multiple versions into V_i. However, multiple in-edges in G capture the multiple choices that we have in recreating V_i from some other versions.

Given graph $G = (V, E)$, the goal of each of our problems is to identify a storage graph $G_s = (V_s, E_s)$, a subset of G, favorably balancing total storage cost and the recreation cost for each version. Implicitly, we will store all versions and deltas corresponding to edges in this storage graph. (We explain this in the context of the example below.) We say a storage graph G_s is *feasible* for a given problem if (a) each version can be recreated based on the information contained or stored in G_s, (b) the recreation cost or the total storage cost meets the constraint listed in each problem.

Example 3: Given matrix Δ and Φ in Figure 2(i) and 2(ii), the corresponding graph G is shown in Figure 3. Every version is reachable from V_0. For example, edge (V_0, V_1) is weighted with $\langle \Delta_{1,1}, \Phi_{1,1} \rangle = \langle 10000, 10000 \rangle$; edge $\langle V_3, V_5 \rangle$ is weighted with $\langle \Delta_{3,5}, \Phi_{3,5} \rangle = \langle 800, 2500 \rangle$. Figure 4 is a feasible storage graph given G in Figure 3, where V_1 and V_3 are materialized (since the edges from V_0 to V_1 and V_3 are present) while V_2, V_4 and V_5 are stored as modifications from other versions.

After mapping our problem into a graph setting, we have the following lemma.

Lemma 1: The optimal storage graph $G_s = (V_s, E_s)$ for all 6 problems listed above must be a spanning tree T rooted at dummy vertex V_0 in graph G.[1]

Recall that a spanning tree is a tree where every vertex is connected and reachable, and has no cycles. For Problems 1 and 2, we have the following observations. A *shortest path tree* is defined as a spanning tree where the path from root to each vertex is a shortest path between those two in the original graph: this would be simply consist of the edges that were explored in an execution of Dijkstra's shortest path algorithm.

Lemma 2: The optimal storage graph G_s for Problem 1 is a minimum spanning tree of Grooted at V_0, considering only weights $\Delta_{i,j}$.

Lemma 3: The optimal storage graph G_s for Problem 2 is a shortest path tree of G rooted at V_0, considering only weights $\Phi_{i,j}$.

[1]We refer the reader to the extended version for proofs.

2.3 ILP Formulation

We present an ILP formulation of the optimization problems described above. Here, we take Problem 6 as an example; other problems are similar. Let $x_{i,j}$ be a binary variable for each edge $(V_i, V_j) \in E$, indicating whether edge (V_i, V_j) is in the storage graph or not. Specifically, $x_{0,j} = 1$ indicates that version V_j is materialized, while $x_{i,j} = 1$ indicates that the modification from version i to version j is stored where $i \neq 0$. Let r_i be a continuous variable for each vertex $V_i \in V$, where $r_0 = 0$; r_i captures the recreation cost for version i (and must be $\leq \theta$).

minimize $\Sigma_{(V_i, V_j) \in E} x_{i,j} \times \Delta_{i,j}$, subject to:

1 $\Sigma_i x_{i,j} = 1, \forall j$

2 $r_j - r_i \geq \Phi_{i,j}$ if $x_{i,j} = 1$

3 $r_i \leq \theta, \forall i$

Lemma 4: Problem 6 is equivalent to the optimization problem described above.

Note however that the general form of an ILP does not permit an if-then statement (as in (2) above). Instead, we can transform to the general form with the aid of a large constant C. Thus, constraint 2 can be expressed as follows:

$$\Phi_{i,j} + r_i - r_j \leq (1 - x_{i,j}) \times C$$

Where C is a "sufficiently large" constant such that no additional constraint is added to the model. For instance, C here can be set as $2*\theta$. On one hand, if $x_{i,j} = 1 \Rightarrow \Phi_{i,j} + r_i - r_j \leq 0$. On the other hand, if $x_{i,j} = 0 \Rightarrow \Phi_{i,j} + r_i - r_j \leq C$. Since C is "sufficiently large", no additional constraint is added.

3. COMPUTATIONAL COMPLEXITY

In this section, we study the complexity of the problems listed in Table 1 under different application scenarios.

Problem 1 and 2 Complexity—As discussed in Section 2, Problem 1 and 2 can be solved in polynomial time by directly applying a minimum spanning tree algorithm (Kruskal's algorithm or Prim's algorithm for undirected graphs; Edmonds' algorithm [35] for directed graphs) and Dijkstra's shortest path algorithm respectively. Kruskal's algorithm has time complexity $O(E \log V)$, while Prim's algorithm also has time complexity $O(E \log V)$ when using binary heap for implementing the priority queue, and $O(E + V \log V)$ when using Fibonacci heap for implementing the priority queue. The running time of Edmonds' algorithm is $O(EV)$ and can be reduced to $O(E + V \log V)$ with faster implementation. Similarly, Dijkstra's algorithm for constructing the shortest path tree starting from the root has a time complexity of $O(E \log V)$ via a binary heap-based priority queue implementation and a time complexity of $O(E + V \log V)$ via Fibonacci heap-based priority queue implementation.

Next, we'll show that Problem 5 and 6 are NP-hard even for the special case where Δ = Φ and Φ is symmetric. This will lead to hardness proofs for the other variants.

Triangle Inequality—The primary challenge that we encounter while demonstrating hardness is that our deltas must obey the triangle inequality: unlike other settings where deltas need not obey real constraints, since, in our case, deltas represent actual modifications that can be stored, it must obey additional realistic constraints. This causes severe complications in proving hardness, often transforming the proofs from very simple to fairly challenging.

Consider the scenario when Δ = Φ and Φ is symmetric. We take Δ as an example. The triangle inequality, can be stated as follows:

$$|\Delta_{p,q}-\Delta_{q,w}| \leq \Delta_{p,w} \leq \Delta_{p,q}+\Delta_{q,w}$$
$$|\Delta_{p,p}-\Delta_{p,q}| \leq \Delta_{q,q} \leq \Delta_{p,p}+\Delta_{p,q}$$

where $p, q, w \in V$ and $p \neq q \neq w$. The first inequality states that the "delta" between two versions can not exceed the total "deltas" of any two-hop path with the same starting and ending vertex; while the second inequality indicates that the "delta" between two versions must be bigger than one version's full storage cost minus another version's full storage cost. Since each tuple and modification is recorded explicitly when Φ is symmetric, it is natural that these two inequalities hold.

Problem 6 Hardness—We now demonstrate hardness.

Lemma 5: Problem 6 is NP-hard when Δ = Φ and Φ is symmetric.

Proof: Here we prove NP-hardness using a reduction from the set cover problem. Recall that in the set cover problem, we are given m sets $S = \{s_1, s_2, \ldots, s_m\}$ and n items $T = \{t_1, t_2, \ldots t_n\}$, where each set s_i covers some items, and the goal is to pick k sets $\mathcal{F} \subset S$ such that $\cup_{\{F \in \mathcal{F}\}} F = T$ while minimizing k.

Given a set cover instance, we now construct an instance of Problem 6 that will provide a solution to the original set cover problem. The threshold we will use in Problem 6 will be $(\beta + 1)\alpha$, where β, α are constants that are each greater than $2(m + n)$. (This is just to ensure that they are "large".) We now construct the graph $G(V, E)$ in the following way; we display the constructed graph in Figure 5. Our vertex set V is as follows:

- $\forall s_i \in S$, create a vertex s_i in V.
- $\forall t_i \in T$, create a vertex t_i in V.
- create an extra vertex v_0, two dummy vertices v_1, v_2 in V.

We add the two dummy vertices simply to ensure that v_0 is materialized, as we will see later. We now define the storage cost for materializing each vertex in V in the following way:

- $\forall s_i \in S$, the cost is α.

- $\forall t_i \in T$, the cost is $(\beta + 1)\alpha$.
- for vertex v_0, the cost is α.
- for vertex v_1, v_2, the cost is $(\beta + 1)\alpha$.

(These are the numbers colored blue in the tree of Figure 5(b).) As we can see above, we have set the costs in such a way that the vertex v_0 and the vertices corresponding to sets in S have low materialization cost, while the other vertices have high materialization cost: this is by design so that we only end up materializing these vertices. Our edge set E is now as follows.

- we connect vertex v_0 to each s_i with weight 1.
- we connect v_0 to both v_1 and v_2 each with weight $\beta\alpha$.
- $\forall s_i \in S$, we connect s_i to t_j with weight $\beta\alpha$ when $t_j \in s_i$, where $\alpha = |V|$.

It is easy to show that our constructed graph G obeys the triangle inequality.

Consider a solution to Problem 6 on the constructed graph G. We now demonstrate that that solution leads to a solution of the original set cover problem. Our proof proceeds in four key steps:

Step 1: The vertex v_0 will be materialized, while v_1, v_2 will not be materialized. Assume the contrary—say v_0 is not materialized in a solution to Problem 6. Then, both v_1 and v_2 must be materialized, because if they are not, then the recreation cost of v_1 and v_2 would be at least $\alpha(\beta + 1) + 1$, violating the condition of Problem 6. However we can avoid materializing v_1 and v_2, instead keep the delta to v_0 and materialize v_0, maintaining the recreation cost as is while reducing the storage cost. Thus v_0 has to be materialized, while v_1, v_2 will not be materialized. (Our reason for introducing v_1, v_2 is precisely to ensure that v_0 is materialized so that it can provide basis for us to store deltas to the sets s_i.)

Step 2: None of the t_i will be materialized. Say a given t_i is materialized in the solution to Problem 6. Then, either we have a set s_j where s_j is connected to t_i in Figure 5(a) also materialized, or not. Let's consider the former case. In the former case, we can avoid materializing t_i, and instead add the delta from s_j to t_i, thereby reducing storage cost while keeping recreation cost fixed. In the latter case, pick any s_j such that s_j is connected to t_i and is not materialized. Then, we must have the delta from v_0 to s_j as part of the solution. Here, we can replace that edge, and materialized t_i, with materialized s_j, and the delta from s_j to t_i: this would reduce the total storage cost while keeping the recreation cost fixed. Thus, in either case, we can improve the solution if any of the t_i are materialized, rendering the statement false.

Step 3: For each s_i, either it is materialized, or the edge from v_0 to s_i will be part of the storage graph. This step is easy to see: since none of the t_i are materialized, either each s_i has to be materialized, or we must store a delta from v_0.

Step 4: The sets s_i that are materialized correspond to a minimal set cover of the original problem. It is easy to see that for each t_j we must have an s_i such that s_i covers t_j, and s_i is materialized, in order for the recreation cost constraint to not be

violated for t_j. Thus, the materialized s_i must be a set cover for the original problem. Furthermore, in order for the storage cost to be as small as possible, as few s_i as possible must be materialized (this is the only place we can save cost). Thus, the materialized s_i also correspond to a minimal set cover for the original problem.

Thus, minimizing the total storage cost is equivalent to minimizing k in set cover problem.

Problem 5 Hardness—We now show that Problem 5 is NP-Hard as well. The general philosophy is similar to the proof in Lemma 5, except that we create c dummy vertices instead of two dummy vertices v_1, v_2 in Lemma 5, where c is sufficiently large—this is to once again ensure that v_0 is materialized. The detailed proof can be found in the extended technical report [13].

Lemma 6: Problem 5 is NP-Hard when $\Delta = \Phi$ and Φ is symmetric.

Since Problem 4 swaps the constraint and goal compared to Problem 6, it is similarly NP-Hard. (Note that the decision versions of the two problems are in fact identical, and therefore the proof still applies.) Similarly, Problem 3 is also NP-Hard. Now that we have proved the NP-hard even in the special case where $\Delta = \Phi$ and Φ is symmetric, we can conclude that Problem 3, 4, 5, 6, are NP-hard in a more general setting where Φ is not symmetric and $\Delta \neq \Phi$, as listed in Table 1.

Hop-Based Variants—In the extended technical report, we also consider the variant of the problem where $\Delta \neq \Phi$ but the recreation cost Φ_{ij} is the same for all pairs of versions, and a version recreation cost is simply the number of *hops* or delta operations to reconstruct the version. The reason why this hop-based variant is interesting is because it is related to a special case of the *d-MinimumSteinerTree* problem, namely the *d-MinimumSpanningTree* problem, i.e., identifying the smallest spanning tree where the diameter is bounded by d. There has been some work on the *d-MinimumSpanningTree* problem [11, 17, 24], including demonstrating hardness for *d-MinimumSpanningTree* (using a reduction from SAT), and also demonstrating hardness of approximation.

Since the hop-based variant is a special case of the last column of Table 1, this indicates that Problem 6 for the most general case is similarly hard to approximate; we suspect similar results hold for the other problems as well. It remains to be seen if hardness of approximation can be demonstrated for the variants in the second and third last columns.

4. PROPOSED ALGORITHMS

As discussed in Section 2, our different application scenarios lead to different problem formulations, spanning different constraints and objectives, and different assumptions about the nature of Φ, Δ.

Given that we demonstrated in the previous section that all the problems are NP-Hard, we focus on developing efficient heuristics. In this section, we present two novel heuristics: first, in Section 4.1, we present LMG, or the Local Move Greedy algorithm, tailored to the case when there is a bound or objective on the *average recreation cost*: thus, this applies to

Problems 3 and 5. Second, in Section 4.2, we present MP, or Modified Prim's algorithm, tailored to the case when there is a bound or objective on the *maximum recreation cost*: thus, this applies to Problems 4 and 6. We present two variants of the MP algorithm tailored to two different settings.

Then, we present two algorithms — in Section 4.3, we present an approximation algorithm called LAST, and in Section 4.4, we present an algorithm called GitH which is based on Git repack. Both of these are adapted from literature to fit our problems and we compare these against our algorithms in Section 5. Note that LAST does not explicitly optimize any objectives or constraints in the manner of LMG, MP, or GitH, and thus the four algorithms are applicable under different settings; LMG and MP are applicable when there is a bound or constraint on the average or maximum recreation cost, while LAST and GitH are applicable when a "good enough" solution is needed. Furthermore, note that all these algorithms apply to both directed and undirected versions of the problems, and to the symmetric and unsymmetric cases.

The pseudocodes for the algorithms can be found in our extended technical report [13].

4.1 Local Move Greedy Algorithm

The LMG algorithm is applicable when we have a bound or constraint on the average case recreation cost. We focus on the case where there is a constraint on the storage cost (Problem 3); the case when there is no such constraint (Problem 5) can be solved by repeated iterations and binary search on the previous problem.

Outline—At a high level, the algorithm starts with the Minimum Spanning Tree (MST) as G_S, and then greedily adds edges from the Shortest Path Tree (SPT) that are not present in G_S, while G_S respects the bound on storage cost.

Detailed Algorithm—The algorithm starts off with G_S equal to the MST. The SPT naturally contains all the edges corresponding to complete versions. The basic idea of the algorithm is to replace deltas in G_S with versions from the SPT that maximize the following ratio:

$$\rho = \frac{\text{reduction in sum of recreation costs}}{\text{increase in storage cost}}$$

This is simply the reduction in total recreation cost per unit addition of weight to the storage graph G_S.

Let ξ consists of edges in the SPT not present in the G_S (these precisely correspond to the versions that are not explicitly stored in the MST, and are instead computed via deltas in the MST). At each "round", we pick the edge $e_{uv} \in \xi$ that maximizes ρ, and replace previous edge $e_{u'v}$ to v. The reduction in the sum of the recreation costs is computed by adding up the reductions in recreation costs of all $w \in G_S$ that are descendants of v in the storage graph (including v itself). On the other hand, the increase in storage cost is simply the weight of

e_{uv} minus the weight of $e_{u'v}$. This process is repeated as long as the storage budget is not violated. We explain this with the means of an example.

Example 4: Figure 6(a) denotes the current G_S. Node 0 corresponds to the dummy node. Now, we are considering replacing edge e_{14} with edge e_{04}, that is, we are replacing a delta to version 4 with version 4 itself. Then, the denominator of ρ is simply $\Delta_{04} - \Delta_{14}$. And the numerator is the changes in recreation costs of versions 4, 5, and 6 (notice that 5 and 6 were below 4 in the tree.) This is actually simple to compute: it is simply three times the change in the recreation cost of version 4 (since it affects all versions equally). Thus, we have the numerator of ρ is simply $3 \times (\Phi_{01} + \Phi_{14} - \Phi_{04})$.

Complexity—Our overall complexity is $O(|V|^2)$. We provide details in the technical report.

Access Frequencies—Note that the algorithm can easily take into account access frequencies of different versions and instead optimize for the total weighted recreation cost (weighted by access frequencies). The algorithm is similar, except that the numerator of ρ will capture the reduction in weighted recreation cost.

4.2 Modified Prim's Algorithm

Next, we introduce a heuristic algorithm based on Prim's algorithm for Minimum Spanning Trees for Problem 6 where the goal is to reduce total storage cost while recreation cost for each version is within threshold θ; the solution for Problem 4 is similar.

Outline—At a high level, the algorithm is a variant of Prim's algorithm, greedily adding the version with smallest storage cost and the corresponding edge to form a spanning tree T. Unlike Prim's algorithm where the spanning tree simply grows, in this case, even if an edge is present in T, it could be removed in future iterations. At all stages, the algorithm maintains the invariant that the recreation cost of all versions in T is bounded within θ.

Detailed Algorithm—At each iteration, the algorithm picks the version V_i with the smallest storage cost to be added to the tree. Once this version V_i is added, we consider adding all deltas to all other versions V_j such that their recreation cost through V_i is within the constraint θ, and the storage cost does not increase. Each version maintains a pair $l(V_i)$ and $d(V_i)$: $l(V_i)$ denotes the marginal storage cost of V_i, while $d(V_i)$ denotes the total recreation cost of V_i. At the start, $l(V_i)$ is simply the storage cost of V_i in its entirety.

We now describe the algorithm in detail. Set X represents the current version set of the current spanning tree T. Initially $X = \emptyset$. In each iteration, the version V_i with the smallest storage cost ($l(V_i)$) in the priority queue PQ is picked and added into spanning tree T. When V_i is added into T, we need to update the storage cost and recreation cost for all V_j that are neighbors of V_i. Notice that in Prim's algorithm, we do not need to consider neighbors that are already in T. However, in our scenario a better path to such a neighbor may be found and this may result in an update. For instance, if edge $\langle V_i, V_j \rangle$ can make V_j 's storage cost smaller while the recreation cost for V_j does not increase, we can update $p(V_j) = V_i$ as well as $d(V_j)$, $l(V_j)$ and T. For neighbors $V_j \notin T$, we update $d(V_j)$, $l(V_j)$, $p(V_j)$ if edge $\langle V_i, V_j \rangle$ can make V_j 's storage cost smaller and the recreation cost for V_j is no bigger than θ.

Example 5: Say we operate on G given by Figure 7, and let the threshold θ be 6. Each version V_i is associated with a pair $\langle l(V_i), d(V_i) \rangle$. Initially version V_0 is pushed into priority queue. When V_0 is dequeued, each neighbor V_j updates $< l(V_j), d(V_j) >$ as shown in Figure 9 (a). Notice that $l(V_i), i \neq 0$ for all i is simply the storage cost for that version. For example, when considering edge (V_0, V_1), $l(V_1) = 3$ and $d(V_1) = 3$ is updated since recreation cost (if V_1 is to be stored in its entirety) is smaller than threshold θ, i.e., $3 < 6$. Afterwards, version V_1, V_2 and V_3 are inserted into the priority queue. Next, we dequeue V_1 since $l(V_1)$ is smallest among the versions in the priority queue, and add V_1 to the spanning tree. We then update $< l(V_j), d(V_j) >$ for all neighbors of V_1, e.g., the recreation cost for version V_2 will be 6 and the storage cost will be 2 when considering edge (V_1, V_2). Since $6 \leq 6$, $(l(V_2), d(V_2))$ is updated to (2, 6) as shown in Figure 9 (b); however, $< l(V_3), d(V_3) >$ will not be updated since the recreation cost is $3 + 4 > 6$ when considering edge (V_1, V_3). Subsequently, version V_2 is dequeued because it has the lowest $l(V_2)$, and is added to the tree, giving Figure 9 (b). Subsequently, version V_3 are dequeued. When V_3 is dequeued from PQ, $(l(V_2), d(V_2))$ is updated. This is because the storage cost for V_2 can be updated to 1 and the recreation cost is still 6 when considering edge (V_3, V_2), even if V_2 is already in T as shown in Figure 9 (c). Eventually, we get the final answer in Figure 9 (d).

Complexity—The complexity of the algorithm is the same as that of Prim's algorithm, i.e., $O(|E| \log |V|)$.

4.3 LAST Algorithm

Here, we sketch an algorithm from previous work [21] that enables us to find a tree with a good balance of storage and recreation costs, under the assumptions that $\Delta = \Phi$ and Φ is symmetric.

Sketch—The algorithm, which takes a parameter α as input, starts with a minimum spanning tree and does a depth-first traveral (DFS) on it. When visiting V_i during the traversal, if it finds that the recreation cost for V_i exceeds $\alpha \times$ the cost of the shortest path from V_0 to V_i, then this current path is replaced with the shortest path to the node. It can be shown that the total cost of the resulting spanning tree is within $(1+2/(\alpha - 1))$ times the cost of minimum spanning tree in G. Even though the algorithm was proposed for undirected graphs, it can be applied to the directed graph case but without any comparable guarantees. We refer the reader to the full version for more details and pseudocode [13].

Example 6: Figure 10 (a) is the minimum spanning tree (MST) rooted at node V_0 of G in Figure 8. The approximation threshold α is set to be 2. The algorithm starts with the MST and conducts a depth-first traversal in the MST from root V_0. When visiting node V_2, $d(V_2) = 3$ and the shortest path to node V_2 is 3, thus $3 < 2 \times 3$. We continue to visit node V_2 and V_3. When visiting V_3, $d(V_3) = 8 > 2 \times 3$ where 3 is the shortest path to V_3 in G. Thus, $d(V_3)$ is set to be 3 and $p(V_3)$ is set to be node 0 by replacing with the shortest path $\langle V_0, V_3 \rangle$ as shown in Figure 10 (b). Afterwards, the back-edge $< V_3, V_1 >$ is traversed in MST. Since $3 + 2 < 6$, where 3 is the current value of $d(V_3)$, 2 is the edge weight of (V_3, V_1) and 6 is the current value in $d(V_1)$, thus $d(V_1)$ is updated as 5 and $p(V_1)$ is updated as node V_3. At last node V_4 is visited, $d(V_4)$ is first updated as 7. Since $7 < 2 \times 4$, lines 9–11 are not executed.

Figure 10 (c) is the resulting spanning tree of the algorithm, where the recreation cost for each node is under the constraint and the total storage cost is 3 + 3 + 2 + 2 = 10.

Complexity—The complexity of the algorithm is $O(|E| \log |V|)$. Further details can be found in the technical report.

4.4 Git Heuristic

This heuristic is an adaptation of the current heuristic used by Git and we refer to it as GitH. We sketch the algorithm here and refer the reader to the extended version for more details [13]. GitH uses two parameters: w (window size) and d (max depth).

We consider the versions in an non-increasing order of their sizes. The first version in this ordering is chosen as the root of the storage graph and has depth 0 (i.e., it is materialized). At all times, we maintain a sliding window containing at most w versions. For each version V_i after the first one, let V_l denote a version in the current window. We compute: $\Delta'_{l,i} = \Delta_{l,i}/(d - d_l)$, where d_l is the depth of V_l (thus deltas with shallow depths are preferred over slightly smaller deltas with higher depths). We find the version V_j with the lowest value of this quantity and choose it as V_i's parent (as long as $d_j < d$). The depth of V_i is then set to $d_j + 1$. The sliding window is modified to move V_l to the end of the window (so it will stay in the window longer), V_j is added to the window, and the version at the beginning of the window is dropped.

Complexity—The running time of the heuristic is $O(|V| \log |V| + w|V|)$, excluding the time to construct deltas.

5. EXPERIMENTS

We have built a prototype version management system, that will serve as a foundation to DataHub [12]. The system provides a subset of Git/SVN-like interface for dataset versioning. Users interact with the version management system in a client-server model over HTTP. The server is implemented in Java, and is responsible for storing the version history of the repository as well as the actual files in them. The client is implemented in Python and provides functionality to create (commit) and check out versions of datasets, and create and merge branches. Note that, unlike traditional VCS which make a best effort to perform automatic merges, in our system we let the user perform the merge and notify the system by creating a version with more than one parent.

Implementation—In the following sections, we present an extensive evaluation of our designed algorithms using a combination of synthetic and derived real-world datasets. Apart from implementing the algorithms described above, LMG and LAST require both SPT and MST as input. For both directed and undirected graphs, we use Dijkstra's algorithm to find the single-source shortest path tree (SPT). We use Prim's algorithm to find the minimum spanning tree for undirected graphs. For directed graphs, we use an implementation [3] of the Edmonds' algorithm [35] for computing the min-cost arborescence (MCA). We ran all

our experiments on a 2.2GHz Intel Xeon CPU E5-2430 server with 64GB of memory, running 64-bit Red Hat Enterprise Linux 6.5.

5.1 Datasets

We use four data sets: two synthetic and two derived from real-world source code repositories. Although there are many publicly available source code repositories with large numbers of commits (e.g., in GitHub), those repositories typically contain fairly small (source code) files, and further the changes between versions tend to be localized and are typically very small; we expect dataset versions generated during collaborative data analysis to contain much larger datasets and to exhibit large changes between versions. We were unable to find any realistic workloads of that kind.

Hence, we generated realistic dataset versioning workloads as follows. First, we wrote a *synthetic version generator suite*, driven by a small set of parameters, that is able to generate a variety of version histories and corresponding datasets. Second, we created two real-world datasets using publicly available forks of popular repositories on GitHub. We describe each of the two below.

Synthetic Datasets—Our synthetic dataset generation suite[2] takes a two-step approach to generate a dataset that we sketch below. The first step is to generate a version graph with the desired structure, controlled by the following parameters:

- number of commits, i.e., the total number of versions.
- branch interval and probability, the number of consecutive versions after which a branch can be created, and probability of creating a branch.
- branch limit, the maximum number of branches from any point in the version history. We choose a number in [1, branch limit] uniformly at random when we decide to create branches.
- branch length, the maximum number of commits in any branch. The actual length is a uniformly chosen integer between 1 and branch length.

Once a version graph is generated, the second step is to generate the appropriate versions and compute the deltas. The files in our synthetic dataset are ordered CSV files (containing tabular data) and we use deltas based on UNIX-style diffs. The previous step also annotates each edge (u, v) in the version graph with edit commands that can be used to produce v from u. Edit commands are a combination of one of the following six instructions – add/delete a set of consecutive rows, add/remove a column, and modify a subset of rows/columns.

Using this, we generated two synthetic datasets (Figure 11):

- **Densely Connected (DC):** This dataset is based on a “flat” version history, i.e., number of branches is high, they occur often and have short lengths. For each version in this data set, we compute the delta with all versions in a 10-hop distance in the version graph to populate additional entries in Δ and Φ.

[2]Our synthetic dataset generator may be of independent interest to researchers working on version management.

- **Linear Chain (LC):** This dataset is based on a "mostly-linear" version history, i.e., number of branches is low, they occur after large intervals and have longer lengths. For each version in this data set, we compute the delta with all versions within a 25-hop distance in the version graph to populate Δ and Φ.

Real-world datasets—We use 986 forks of the Twitter Bootstrap repository and 100 forks of the Linux repository, to derive our real-world workloads. For each repository, we checkout the latest version in each fork and concatenate all files in it (by traversing the directory structure in lexicographic order). Thereafter, we compute deltas between all pairs of versions in a repository, provided the size difference between the versions under consideration is less than a threshold. We set this threshold to 100KB for the Twitter Bootstrap repository and 10MB for the Linux repository. This gives us two real-world datasets, Bootstrap Forks (BF) and Linux Forks (LF), with properties shown in Figure 11.

5.2 Comparison with SVN and Git

We begin with evaluating the performance of two popular version control systems, SVN (v1.8.8) and Git (v1.7.1), using the LF dataset. We create an FSFS-type repository in SVN, which is more space efficient than a Berkeley DB-based repository [4]. We then import the entire LF dataset into the repository in a single commit. The amount of space occupied by the db/revs/directory is around 8.5GB and it takes around 48 minutes to complete the import. We contrast this with the naive approach of applying a gzip on the files which results in total compressed storage of 10.2GB. In case of Git, we add and commit the files in the repository and then run a git repack -a -d –depth=50 –window=50 on the repository[3]. The size of the Git pack file is 202 MB although the repack consumes 55GB memory and takes 114 minutes (for higher window sizes, Git fails to complete the repack as it runs out of memory).

In comparison, the solution found by the MCA algorithm occupies 516MB of compressed storage (2.24GB when uncompressed) when using UNIX diff for computing the deltas. To make a fair comparison with Git, we use xdiff from the LibXDiff library [7] for computing the deltas, which forms the basis of Git's delta computing routine. Using xdiff brings down the total storage cost to just 159 MB. The total time taken is around 102 minutes; this includes the time taken to compute the deltas and then to find the MCA for the corresponding graph.

The main reason behind SVN's poor performance is its use of "skip-deltas" to ensure that at most $O(\log n)$ deltas are needed for reconstructing any version [8]; that tends to lead it to repeatedly store redundant delta information as a result of which the total space requirement increases significantly. The heuristic used by Git is much better than SVN (Section 4.4). However as we show later (Fig. 12), our implementation of that heuristic (GitH) required more storage than LMG for guaranteeing similar recreation costs.

[3]Unlike git repack, svnadmin pack has a negligible effect on the storage cost as it primarily aims to reduce disk seeks and per-version disk usage penalty by concatenating files into a single "pack" [5, 6].

5.3 Experimental Results

Directed Graphs—We begin with a comprehensive evaluation of the three algorithms, LMG, MP, and LAST, on directed datasets. Given that all of these algorithms have parameters that can be used to trade off the storage cost and the total recreation cost, we compare them by plotting the different solutions they are able to find for the different values of their respective input parameters. Figure 12(a–d) show four such plots; we run each of the algorithms with a range of different values for its input parameter and plot the storage cost and the total (sum) recreation cost for each of the solutions found. We also show the minimum possible values for these two costs: the vertical dashed red line indicates the minimum storage cost required for storing the versions in the dataset as found by MCA, and the horizontal one indicates the minimum total recreation cost as found by SPT (equal to the sum of all version sizes).

The first key observation we make is that, the total recreation cost decreases drastically by allowing a small increase in the storage budget over MCA. For example, for the DC dataset, the sum recreation cost for MCA is over 11 PB (see Table 11) as compared to just 34TB for the SPT solution (which is the minimum possible). As we can see from Figure 12(a), a space budget of 1.1×the MCA storage cost reduces the sum of recreation cost by three orders of magnitude. Similar trends can be observed for the remaining datasets and across all the algorithms. We observe that LMG results in the best tradeoff between the sum of recreation cost and storage cost with LAST performing fairly closely. **An important takeaway here, especially given the amount of prior work that has focused purely on storage cost minimization (Section 6), is that: it is possible to construct balanced trees where the sum of recreation costs can be reduced and brought close to that of SPT while using only a fraction of the space that SPT needs.**

We also ran GitH heuristic on the all the four datasets with varying window and depth settings. For BF, we ran the algorithm with four different window sizes (50, 25, 20, 10) for a fixed depth 10 and provided the GitH algorithm with all the deltas that it requested. For all other datasets, we ran GitH with an infinite window size but restricted it to choose from deltas that were available to the other algorithms (i.e., only deltas with sizes below a threshold); as we can see, the solutions found by GitH exhibited very good total recreation cost, but required significantly higher storage than other algorithms. This is not surprising given that GitH is a greedy heuristic that makes choices in a somewhat arbitrary order.

In Figures 13(a–b), we plot the maximum recreation costs instead of the sum of recreation costs across all versions for two of the datasets (the other two datasets exhibited similar behavior). The MP algorithm found the best solutions here for all datasets, and we also observed that LMG and LAST both show plateaus for some datasets where the maximum recreation cost did not change when the storage budget was increased. This is not surprising given that the basic MP algorithm tries to optimize for the storage cost given a bound on the maximum recreation cost, whereas both LMG and LAST focus on minimization of the storage cost and one version with high recreation cost is unlikely to affect that significantly.

Undirected Graphs—We test the three algorithms on the undirected versions of three of the datasets (Figure 14). For DC and LC, undirected deltas between pairs of versions were

obtained by concatenating the two directional deltas; for the BF dataset, we use UNIX diff itself to produce undirected deltas. Here again we observe that LMG consistently outperforms the other algorithms in terms of finding a good balance between the storage cost and the sum of recreation costs. MP again shows the best results when trying to balance the maximum recreation cost and the total storage cost. Similar results were observed for other datasets but are omitted due to space limitations.

Workload-aware Sum of Recreation Cost Optimization—In many cases, we may be able to estimate access frequencies for the various versions (from historical access patterns), and if available, we may want to take those into account when constructing the storage graph. The LMG algorithm can be easily adapted to take such information into account, whereas it is not clear how to adapt either LAST or MP in a similar fashion. In this experiment, we use LMG to compute a storage graph such that the sum of recreation costs is minimal given a space budget, while taking workload information into account. The worload here assigns a frequency of access to each version in the repository using a Zipfian distribution (with exponent 2); real-world access frequencies are known to follow such distributions. Given the workload information, the algorithm should find a storage graph that has the sum of recreation cost less than the index when the workload information is not taken into account (i.e., all versions are assumed to be accessed equally frequently). Figure 15 shows the results for this experiment. As we can see, for the DC dataset, taking into account the access frequencies during optimization led to much better solutions than ignoring the access frequencies. On the other hand, for the LF dataset, we did not observe a large difference.

Running Times—Here we evaluate the running times of the LMG algorithm. Recall that LMG takes MST (or MCA) and SPT as inputs. In Fig. 16, we report the total running time as well as the time taken by LMG itself. We generated a set of version graphs as subsets of the graphs for LC and DC datasets as follows: for a given number of versions n, we randomly choose a node and traverse the graph starting at that node in breadth-first manner till we construct a subgraph with n versions. We generate 5 such subgraphs for increasing values of n and report the average running time for LMG; the storage budget for LMG is set to three times of the space required by the MST (all our reported experiments with LMG use less storage budget than that). The time taken by LMG on DC dataset is more than LC for the same number of versions; this is because DC has lower delta values than LC (see Fig. 11) and thus requires more edges from SPT to satisfy the storage budget.

On the other hand, MP takes between 1 to 8 seconds on those datasets, when the recreation cost is set to maximum. Similar to LMG, LAST requires the MST/MCA and SPT as inputs; however the running time of LAST itself is linear and it takes less than 1 second in all cases. Finally the time taken by GitH on LC and DC datasets, on varying window sizes range from 35 seconds (window = 1000) to a little more than 120 minutes (window = 100000); note that, this excludes the time for constructing the deltas.

In summary, although LMG is inherently a more expensive algorithm than MP or LAST, it runs in reasonable time on large input sizes; we note that all of these times are likely to be dwarfed by the time it takes to construct deltas even for moderately-sized datasets.

Comparison with ILP solutions—Finally, we compare the quality of the solutions found by MP with the optimal solution found using the Gurobi Optimizer for Problem 6. We use the ILP formulation from Section 2.3 with constraint on the maximum recreation cost (θ), and compare the optimal storage cost with that of the MP algorithm (which resulted in solutions with lowest maximum recreation costs in our evaluation). We use our synthetic dataset generation suite to generate three small datasets, with 15, 25 and 50 versions denoted by v15, v25 and v50 respectively and compute deltas between all pairs of versions. Table 2 reports the results of this experiment, across five θ values. The ILP turned out to be very difficult to solve, even for the very small problem sizes, and in many cases, the optimizer did not finish and the reported numbers are the best solutions found by it.

As we can see, the solutions found by MP are quite close to the ILP solutions for the small problem sizes for which we could get any solutions out of the optimizer. However, extrapolating from the (admittedly limited) data points, we expect that on large problem sizes, MP may be significantly worse than optimal for some variations on the problems (we note that the optimization problem formulations involving max recreation cost are likely to turn out to be harder than the formulations that focus on the average recreation cost). Development of better heuristics and approximation algorithms with provable guarantees for the various problems that we introduce are rich areas for further research.

6. RELATED WORK

Perhaps the most closely related prior work is source code version systems like Git, Mercurial, SVN, and others, that are widely used for managing source code repositories. Despite their popularity, these systems largely use fairly simple algorithms underneath that are optimized to work with modest-sized source code files and their on-disk structures are optimized to work with line-based diffs. These systems are known to have significant limitations when handling large files and large numbers of versions [2]. As a result, a variety of extensions like git-annex [9], git-bigfiles [10], etc., have been developed to make them work reasonably well with large files.

There is much prior work in the temporal databases literature [14, 31, 26, 34] on managing a linear chain of versions, and retrieving a version as of a specific time point (called *snapshot* queries) [29]. [15] proposed an archiving technique where all versions of the data are merged into one hierarchy. It was not, however, a full-fledged version control system representing an arbitrarily graph of versions. Snapshot queries have recently also been studied in the context of array databases [32, 30] and graph databases [22]. Seering et al. [30] proposed an MST-like technique for storing an arbitrary tree of versions in the context of scientific databases. They also proposed several heuristics for choosing which versions to materialize given the distribution of access frequencies to historical versions. Several databases support “time travel” features (e.g., Oracle Flashback, Postgres [33]). However, those do not allow for branching trees of versions. [19] articulates a similar vision to our overall DataHub vision; however, they do not propose formalisms or algorithms to solve the underlying data management challenges.

There is also much prior work on compactly encoding differences between two files or strings in order to reduce communication or storage costs. In addition to standard utilities like UNIX diff, many sophisticated techniques have been proposed for computing differences or edit sequences between two files (e.g., xdelta [25], vdelta [20], vcdiff [23], zdelta [36]). That work is largely orthogonal and complementary to our work.

Many prior efforts have looked at the problem of minimizing the total storage cost for storing a collection of related files (i.e., Problem 1). These works do not typically consider the recreation cost or the tradeoffs between the two. Quinlan et al. [28] propose an archival "deduplication" storage system that identifies duplicate blocks across files and only stores them once for reducing storage requirements. Zhu et al. [37] present several optimizations on the basic theme. Douglis et al. [18] present several techniques to identify pairs of files that could be efficiently stored using delta compression even if there is no explicit derivation information known about the two files; similar techniques could be used to better identify which entries of the matrices Δ and Φ to reveal in our scenario. Burns and Long [16] present a technique for in-place reconstruction of delta-compressed files using a graph-theoretic approach. That work could be incorporated into our overall framework to reduce the memory requirements during reconstruction. We refer the reader to a recent survey [27] for a more comprehensive coverage of this line of work.

7. CONCLUSIONS AND FUTURE WORK

Large datasets and collaborative and iterative analysis are becoming a norm in many application domains; however we lack the data management infrastructure to efficiently manage such datasets, their versions over time, and derived data products. Given the high overlap and duplication among the datasets, it is attractive to consider using delta compression to store the datasets in a compact manner, where some datasets or versions are stored as modifications from other datasets; such delta compression however leads to higher latencies while retrieving specific datasets. In this paper, we studied the trade-off between the storage and recreation costs in a principled manner, by formulating several optimization problems that trade off these two in different ways and showing that most variations are NP-Hard. We also presented several efficient algorithms that are effective at exploring this trade-off, and we presented an extensive experimental evaluation using a prototype version management system that we have built. There are many interesting and rich avenues for future work that we are planning to pursue. In particular, we plan to develop online algorithms for making the optimization decisions as new datasets or versions are being created, and also adaptive algorithms that reevaluate the optimization decisions based on changing workload information. We also plan to explore the challenges in extending our work to a distributed and decentralized setting.

Acknowledgments

This research was supported by NSF Grants IIS-1319432 and IIS-1513407, grant 1U54GM114838 awarded by NIGMS through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative, a Google Faculty Research Award, and an IBM Faculty Award.

References

1. http://git.kernel.org/cgit/git/git.git/tree/Documentation/technical/pack-heuristics.txt.
2. http://comments.gmane.org/gmane.comp.version-control.git/189776.
3. http://edmonds-alg.sourceforge.net/.
4. http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs.
5. http://svnbook.red-bean.com/en/1.8/svn.reposadmin.maint.html#svn.reposadmin.maint.diskspace.fsfspacking.
6. http://svn.apache.org/repos/asf/subversion/trunk/notes/fsfs-improvements.txt.
7. http://www.xmailserver.org/xdiff-lib.html.
8. http://svn.apache.org/repos/asf/subversion/trunk/notes/skip-deltas.
9. https://git-annex.branchable.com/.
10. http://caca.zoy.org/wiki/git-bigfiles.
11. Bar-Ilan J, Kortsarz G, Peleg D. Generalized submodular cover problems and applications. Theoretical Computer Science. 2001; 250(1):179–200.
12. Bhardwaj A, Bhattacherjee S, Chavan A, Deshpande A, Elmore A, Madden S, Parameswaran A. DataHub: Collaborative data science & dataset version management at scale. CIDR. 2015
13. Bhattacherjee S, Chavan A, Huang S, Deshpande A, Parameswaran A. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. ArXiv e-prints. May.2015
14. Bolour A, Anderson TL, Dekeyser LJ, Wong HKT. The role of time in information processing: a survey. SIGMOD Rec. 1982
15. Buneman P, Khanna S, Tajima K, Tan WC. Archiving scientific data. ACM Transactions on Database Systems (TODS). 2004; 29(1):2–42.
16. Burns R, Long D. In-place reconstruction of delta compressed files. In. PODC. 1998
17. Charikar M, Chekuri C, Cheung T-y, Dai Z, Goel A, Guha S, Li M. Approximation algorithms for directed steiner problems. J Alg. 1999
18. Douglis F, Iyengar A. Application-specific delta-encoding via resemblance detection. USENIX ATC. 2003
19. Gatterbauer W, Suciu D. Managing structured collections of community data. CIDR. 2011
20. Hunt J, Vo K, Tichy W. Delta algorithms: An empirical analysis. ACM Trans Softw Eng Methodol. 1998
21. Khuller S, Raghavachari B, Young N. Balancing minimum spanning trees and shortest-path trees. Algorithmica. 1995; 14(4):305–321.
22. Khurana U, Deshpande A. Efficient snapshot retrieval over historical graph data. ICDE. 2013:997–1008.
23. Korn D, Vo K. Engineering a differencing and compression data format. USENIX ATC. 2002
24. Kortsarz G, Peleg D. Approximating shallow-light trees. SODA. 1997
25. MacDonald, J. PhD thesis. UC Berkeley; 2000. File system support for delta compression.
26. Ozsoyoglu G, Snodgrass R. Temporal and real-time databases: a survey. IEEE TKDE. Aug; 1995 7(4):513–532.
27. Paulo J, Pereira J. A survey and classification of storage deduplication systems. ACM Comput Surv. Jun; 2014 47(1):11:1–11:30.
28. Quinlan S, Dorward S. Venti: A new approach to archival storage. FAST. 2002
29. Salzberg B, Tsotras V. Comparison of access methods for time-evolving data. ACM Comput Surv. 1999; 31(2)
30. Seering A, Cudre-Mauroux P, Madden S, Stonebraker M. Efficient versioning for scientific array databases. ICDE. 2012
31. Snodgrass R, Ahn I. A Taxonomy of Time in Databases. SIGMOD. 1985
32. Soroush E, Balazinska M. Time travel in a scientific array database. ICDE. 2013:98–109.
33. Stonebraker M, Kemnitz G. The Postgres next generation database management system. Communications of the ACM. 1991; 34(10):78–92.

34. Tansel, A.Clifford, J.Gadia, S.Jajodia, S.Segev, A., RS, editors. Temporal Databases: Theory, Design, and Implementation. 1993.
35. Tarjan RE. Finding optimum branchings. Networks. 1977; 7(1):25–35.
36. Trendafilov D, Memon N, Suel T. zdelta: An efficient delta compression tool. Technical report. 2002
37. Zhu B, Li K, Patterson R. Avoiding the disk bottleneck in the data domain deduplication file system. FAST. 2008

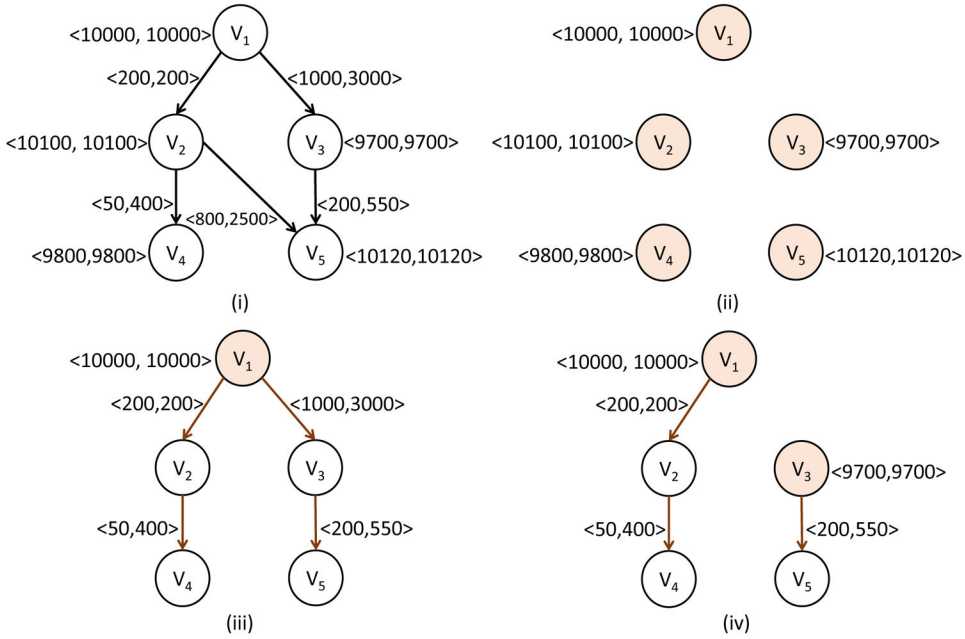

Figure 1.

(i) A version graph over 5 datasets – annotation $\langle a, b \rangle$ indicates a storage cost of a and a recreation cost of b; (ii, iii, iv) three possible storage graphs

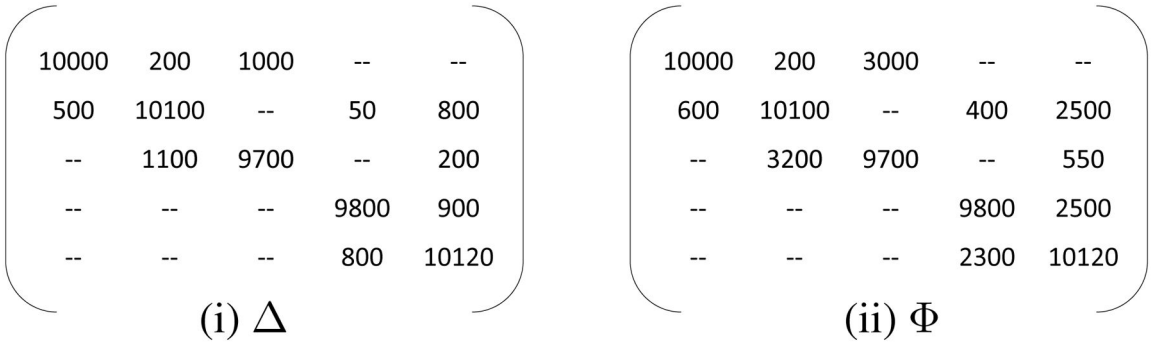

Figure 2.

Matrices corresponding to the example in Figure 1 (with additional entries revealed beyond the ones given by version graph)

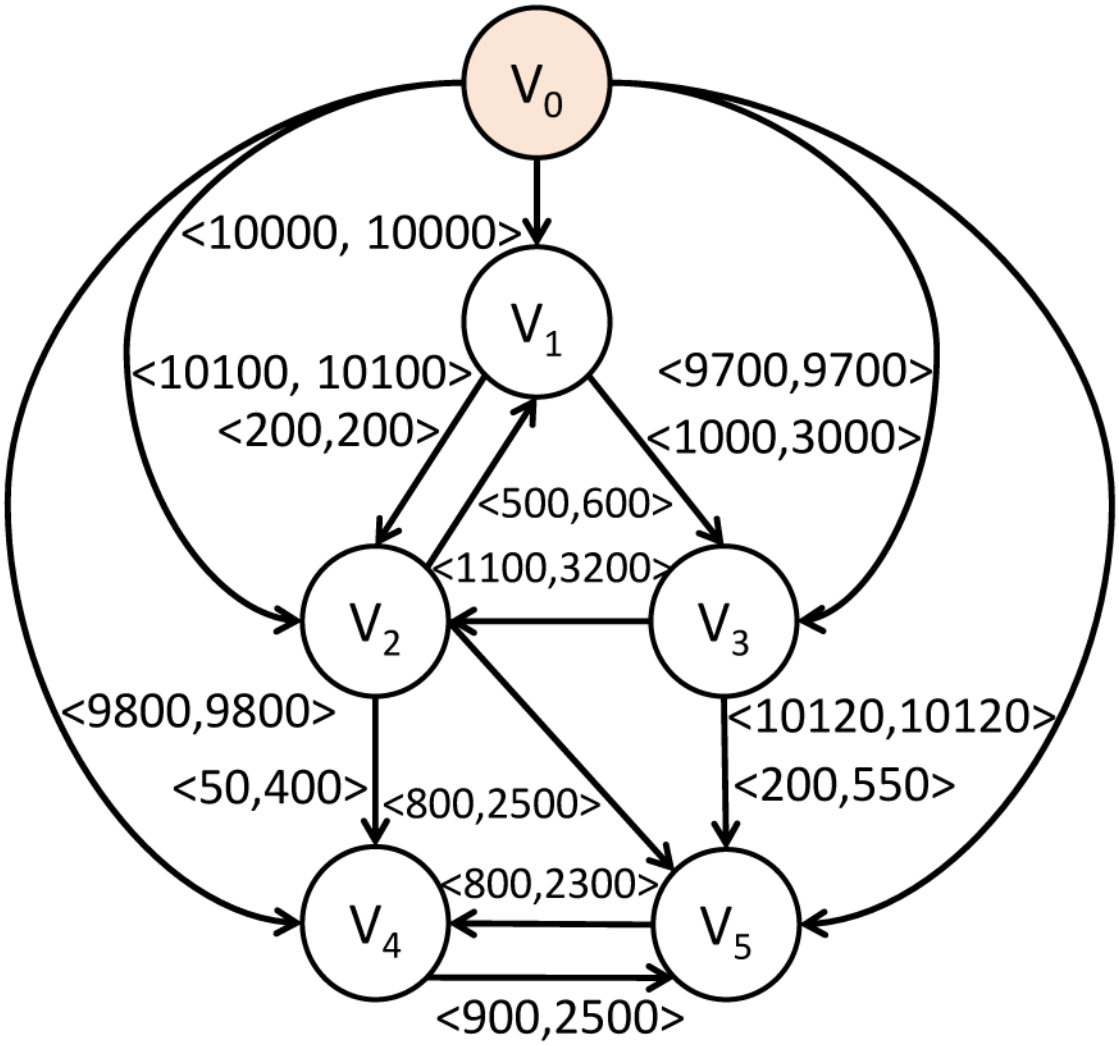

Figure 3.
Graph *G*

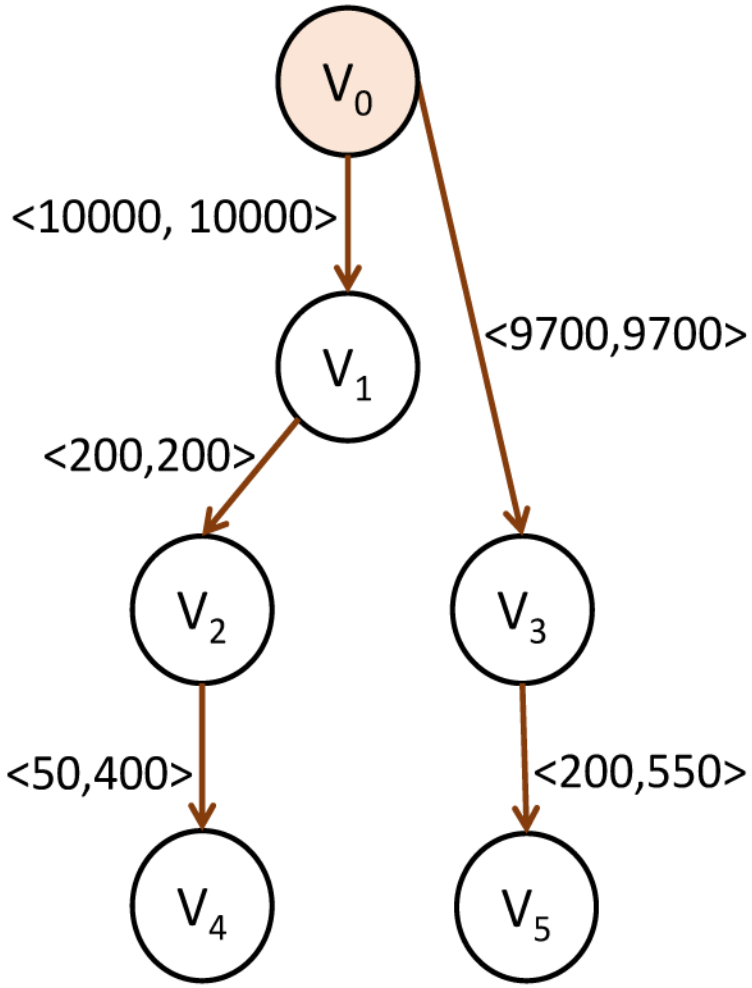

Figure 4.
Storage Graph G_s

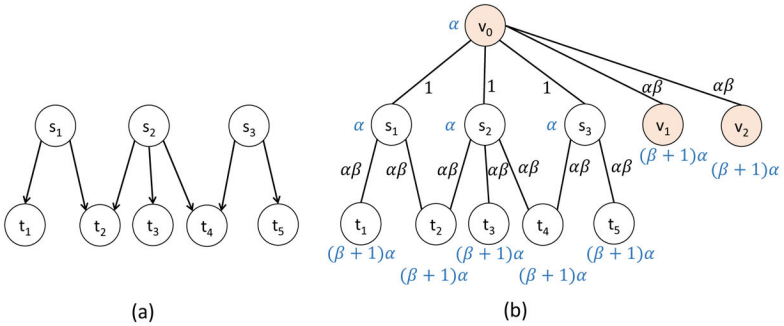

Figure 5.
Illustration of Proof of Lemma 5

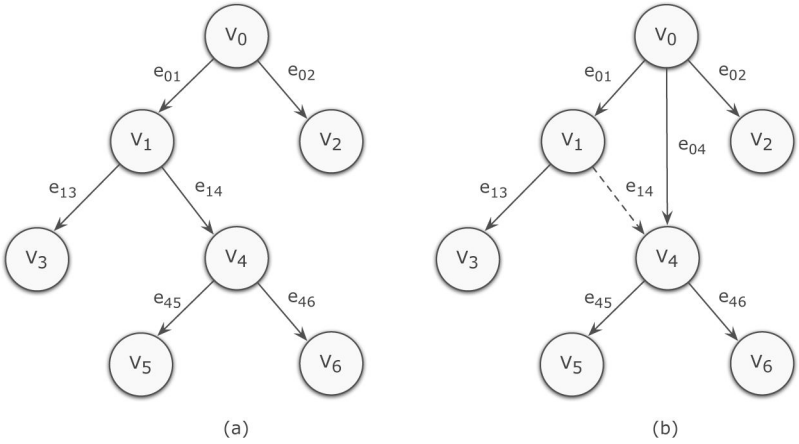

Figure 6.
Illustration of Local Move Greedy Heuristic

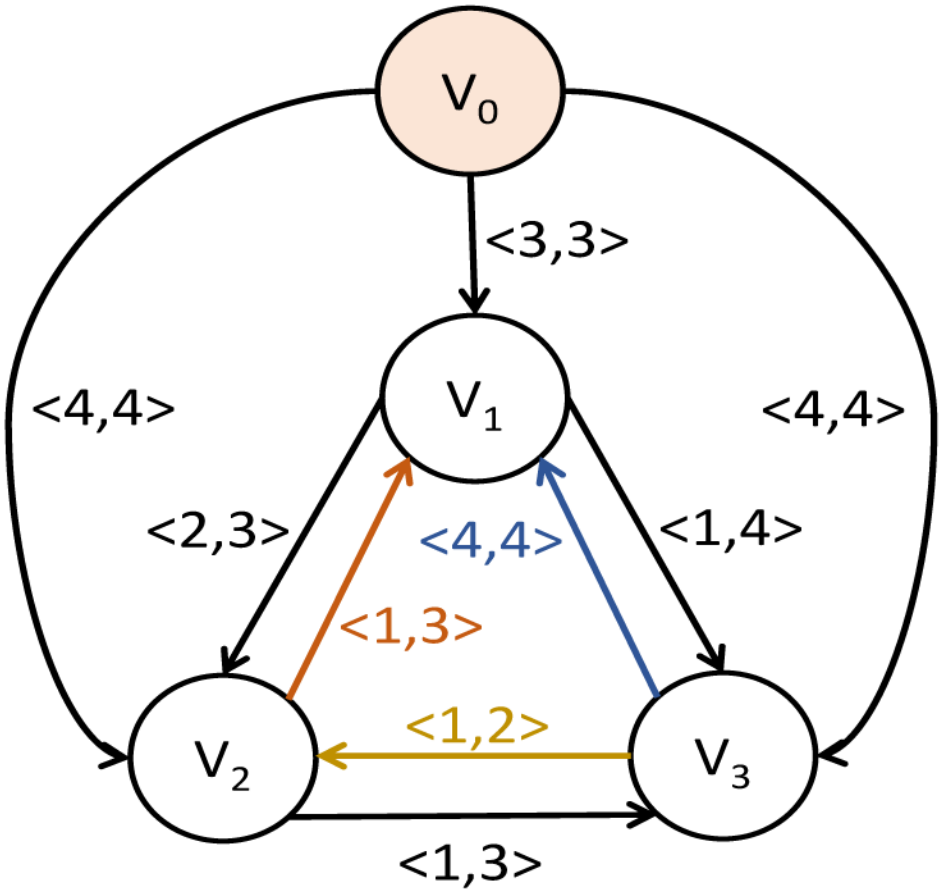

Figure 7.
Directed Graph *G*

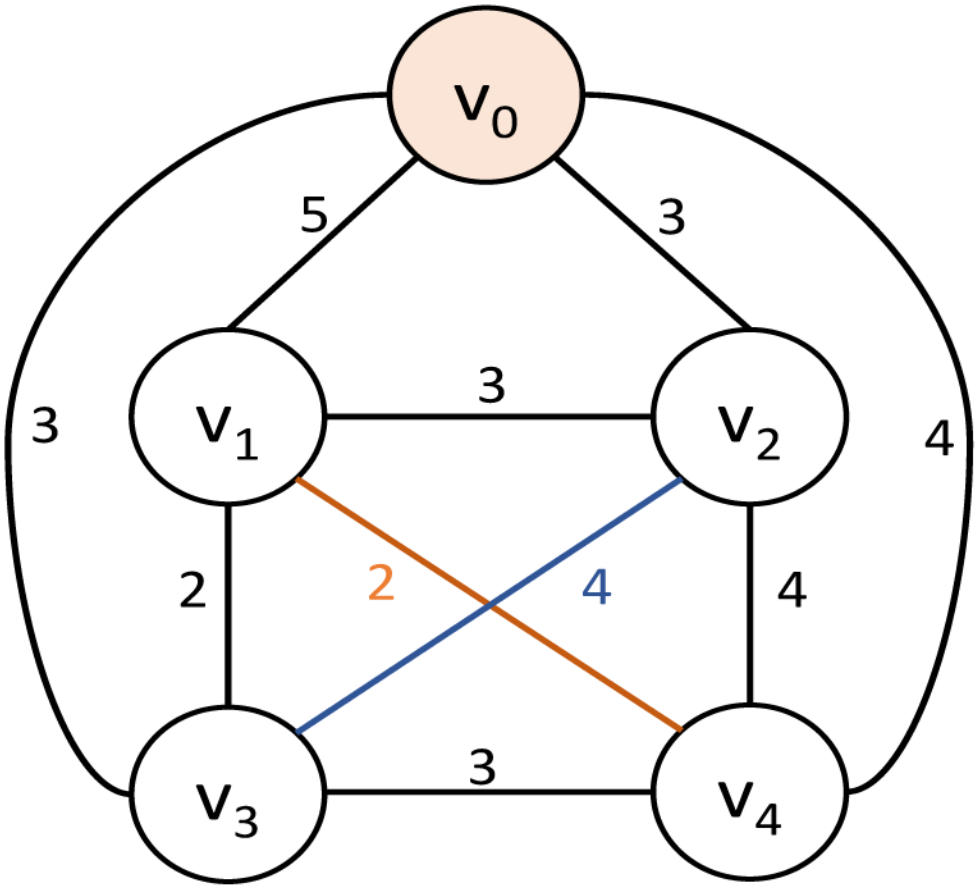

Figure 8.
Undirected Graph *G*

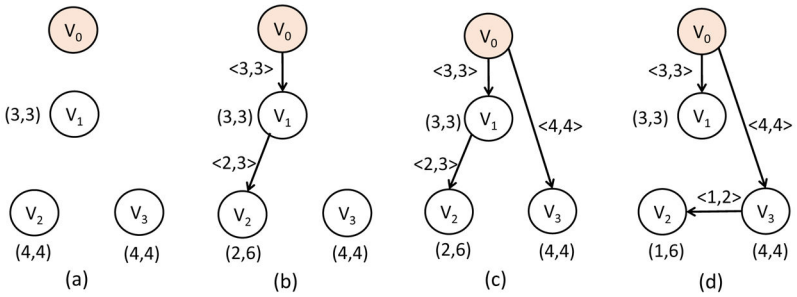

Figure 9.
Illustration of Modified Prim's algorithm in Figure 7

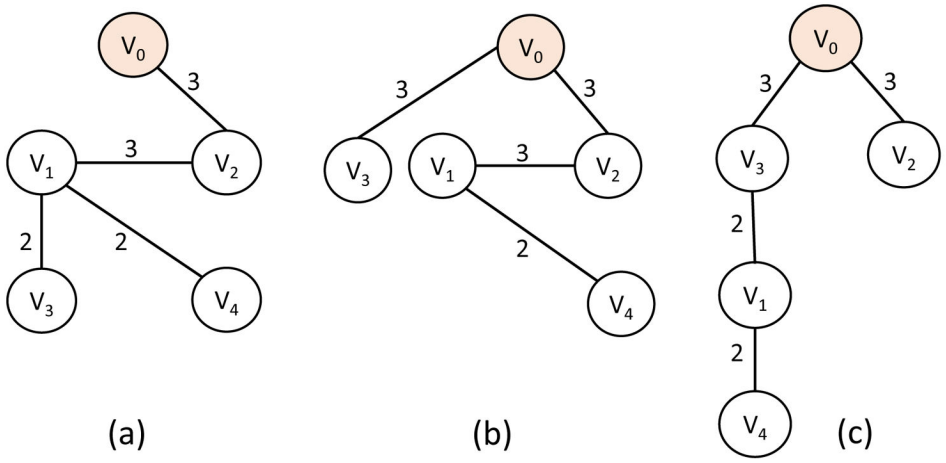

Figure 10.
Illustration of LAST on Figure 8

Dataset	DC	LC	BF	LF
Number of versions	100010	100002	986	100
Number of deltas	18086876	2916768	442492	3562
Average version size (MB)	347.65	356.46	0.401	422.79
MCA-Storage Cost (GB)	1265.34	982.27	0.0250	2.2402
MCA-Sum Recreation Cost (GB)	11506437.83	29934960.95	0.9648	47.6046
MCA-Max Recreation Cost (GB)	257.6	717.5	0.0063	0.5998
SPT-Storage Cost (GB)	33953.84	34811.14	0.3854	41.2881
SPT-Sum Recreation Cost (GB)	33953.84	34811.14	0.3854	41.2881
SPT-Max Recreation Cost (GB)	0.524	0.55	0.0063	0.5091

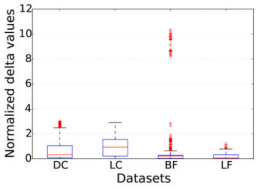

Figure 11.

Dataset properties and distribution of delta sizes (each delta size scaled by the average version size in the dataset).

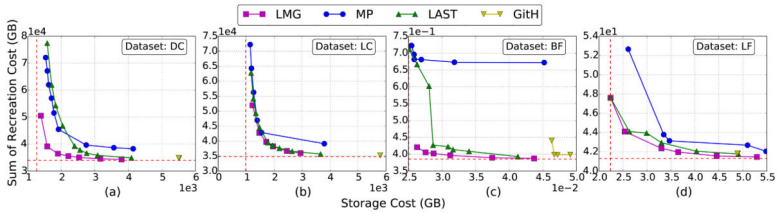

Figure 12.
Results for the directed case, comparing the storage costs and total recreation costs

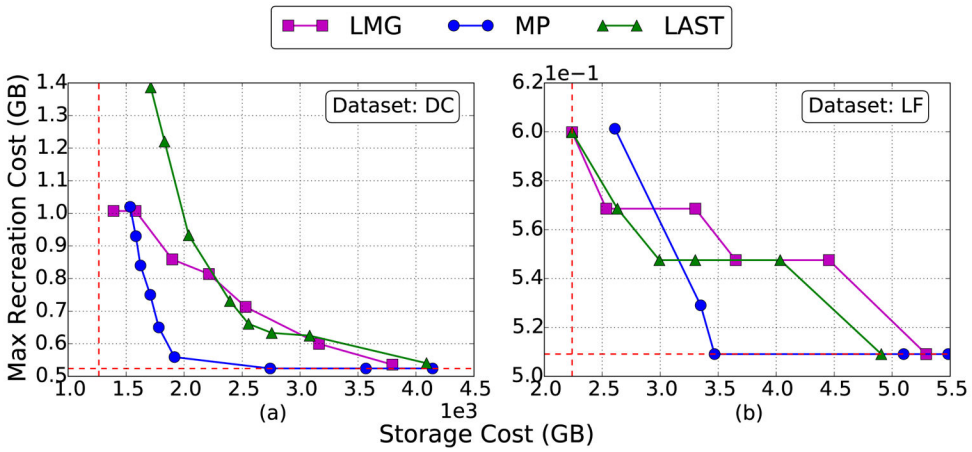

Figure 13.
Results for the directed case, comparing the storage costs and maximum recreation costs

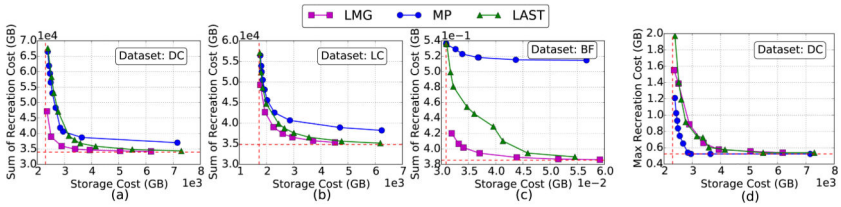

Figure 14.

Results for the undirected case, comparing the storage costs and total recreation costs (a–c) or maximum recreation costs (d)

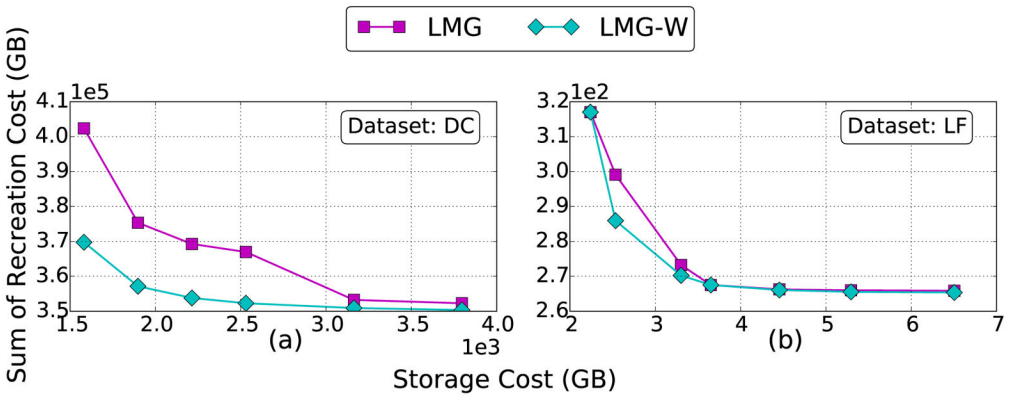

Figure 15.
Taking workload into account leads to better solutions

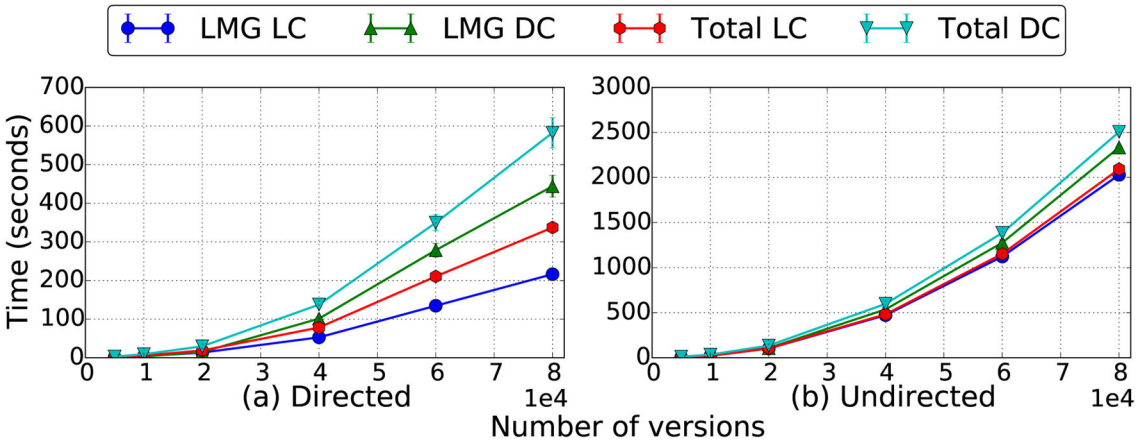

Figure 16.
Running times of LMG

Table 1

Problem Variations With Different Constraints, Objectives and Scenarios.

	Storage Cost	**Recreation Cost**	**Undirected Case, Δ = Φ**	**Directed Case, Δ = Φ**	**Directed Case, Δ ≠ Φ**
Problem 1	minimize $\{\mathscr{C}\}$	$\mathscr{R}_i < \infty, \forall i$	PTime, Minimum Spanning Tree		
Problem 2	$\mathscr{C} < \infty$	minimize $\{\max\{\mathscr{R}_i / 1 \leq i \leq n\}\}$	PTime, Shortest Path Tree		
Problem 3	$\mathscr{C} \leq \beta$	$\text{minimize}\{\sum_{i=1}^{n} \mathscr{R}_i\}$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
Problem 4	$\mathscr{C} \leq \beta$	minimize $\{\max\{\mathscr{R}_i / 1 \leq i \leq n\}\}$		NP-hard, MP Algorithm	
Problem 5	minimize $\{\mathscr{C}\}$	$\sum_{i=1}^{n} \mathscr{R}_i \leq \theta$	NP-hard, LAST Algorithm†	NP-hard, LMG Algorithm	
Problem 6	minimize $\{\mathscr{C}\}$	$\max\{\mathscr{R}_i / 1 \leq i \leq n\} \leq \theta$		NP-hard, MP Algorithm	

Proceedings VLDB Endowment. Author manuscript; available in PMC 2017 July 25.

Table 2

Comparing ILP and MP solutions for small datasets, given a bound on max recreation cost, θ (in GB)

		Storage Cost (GB)				
v15	θ	0.20	0.21	0.22	0.23	0.24
	ILP	0.36	0.36	0.22	0.22	0.22
	MP	0.36	0.36	0.23	0.23	0.23
v25	θ	0.63	0.66	0.69	0.72	0.75
	ILP	2.39	1.95	1.50	1.18	1.06
	MP	2.88	2.13	1.7	1.18	1.18
v50	θ	0.30	0.34	0.41	0.54	0.68
	ILP	1.43	1.10	0.83	0.66	0.60
	MP	1.59	1.45	1.06	0.91	0.82

Proceedings VLDB Endowment. Author manuscript; available in PMC 2017 July 25.